MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 1, 2020

journal homepage: www.mais-journal.ru

SOFTWARE

Parallel Algorithm for Solving the Graph Isomorphism Problem

V. V. Vasilchikov[1]

DOI: 10.18255/1818-1015-2020-1-86-94

[1]P. G. Demidov Yaroslavl State University, 14 Sovetskaya, Yaroslavl 150003, Russia.

MSC2020: 68W10
Research article
Full text in Russian

Received January 16, 2020
After revision February 24, 2020
Accepted February 28, 2020

In this paper, we offer an efficient parallel algorithm for solving the Graph Isomorphism Problem. Our goal is to construct a suitable vertex substitution or to prove the absence of such. The problem is solved for undirected graphs without loops and multiple edges, it is assumed that the graphs can be disconnected. The question of the existence or absence of an algorithm for solving this problem with polynomial complexity is currently open. Therefore, as for any time-consuming task, the question arises of accelerating its solution by parallelizing the algorithm. We used the RPM_ParLib library developed by the author as the main tool to program the algorithm. This library allows us to develop effective applications for parallel computing on a local network in the .NET Framework. Such applications have the ability to generate parallel branches of computation directly during program execution and dynamically redistribute work between computing modules. Any language with support for the .NET Framework can be used as a programming language in conjunction with this library. For our experiments, we developed some C# applications using this library. The main purpose of these experiments was to study the acceleration achieved by recursive-parallel computing. Specially generated random regular graphs with varying degrees of vertices were used as initial data. A detailed description of the algorithm and its testing, as well as the results obtained, are also given in the paper.

Keywords: graph isomorphism problem; parallel algorithm; recursion; .NET

INFORMATION ABOUT THE AUTHORS

Vladimir Vasilyevich Vasilchikov | orcid.org/0000-0001-7882-8906. E-mail: vvv193@mail.ru
PhD.

Funding: This work was supported by initiative program VIP-004 (state registration number AAAA-A16-116070610022-6).

For citation: V. V. Vasilchikov, “Parallel Algorithm for Solving the Graph Isomorphism Problem”, *Modeling and analysis of information systems*, vol. 27, no. 1, pp. 86-94, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 1, 2020
сайт журнала: www.mais-journal.ru

SOFTWARE

Параллельный алгоритм решения задачи об изоморфизме графов

В. В. Васильчиков[1]

DOI: 10.18255/1818-1015-2020-1-86-94

[1]Ярославский государственный университет им. П. Г. Демидова, ул. Советская, 14, Ярославль, 150003 Россия.

УДК 519.688: 519.85
Научная статья
Полный текст на русском языке

Получена 16 января 2020 г.
После доработки 24 февраля 2020 г.
Принята к публикации 28 февраля 2020 г.

В данной работе предлагается параллельный алгоритм решения задачи об изоморфизме графов. Целевым результатом для нас выступает построение подходящей подстановки вершин, либо доказательство отсутствия таковой. Задача решается для неориентированных графов без петель и кратных ребер, допускается, что графы могут быть несвязными. Вопрос о существовании либо отсутствии алгоритма с полиномиальной трудоемкостью в настоящее время является открытым. Следовательно, как и для любой трудоемкой задачи, возникает вопрос об ускорении ее решения за счет распараллеливания алгоритма. Для организации параллельных вычислений автором использовалась библиотека RPM_ParLib, которая позволяет создавать параллельные приложения, работающие в локальной вычислительной сети под управлением среды исполнения .NET Framework. Библиотека поддерживает рекурсивно-параллельный стиль программирования и обеспечивает эффективное распределение работы и динамическую балансировку загрузки вычислительных модулей в процессе исполнения программы. Она может быть использована для приложений, написанных на любом языке программирования, поддерживаемом .NET Framework. Для решения нашей задачи и проведения численного эксперимента было разработано несколько приложений на языке C#. Целью эксперимента было исследование ускорения, достигаемого за счет рекурсивно-параллельной организации вычислений. В качестве исходных данных использовались специально сгенерированные случайные регулярные графы с различной степенью вершин. Подробное описание алгоритма и эксперимента, а также полученные результаты также приводятся в работе.

Ключевые слова: изоморфизм графов; параллельный алгоритм; рекурсия; .NET

ИНФОРМАЦИЯ ОБ АВТОРАХ

Владимир Васильевич Васильчиков | orcid.org/0000-0001-7882-8906. E-mail: vvv193@mail.ru
канд. техн. наук, зав. кафедрой вычислительных и программных систем.

Финансирование: Работа выполнена в рамках инициативной НИР ВИП-004
(номер госрегистрации АААА-А16-116070610022-6).

Для цитирования: V. V. Vasilchikov, “Parallel Algorithm for Solving the Graph Isomorphism Problem”, *Modeling and analysis of information systems*, vol. 27, no. 1, pp. 86-94, 2020.

Введение

Задача об изоморфизме графов является одной из классических задач дискретной оптимизации [1]. Для нее пока не доказана принадлежность ни к классу P, ни к классу NP-полных задач. Потребность в решении этой задачи возникает в самых разных предметных областях, где требуется установление идентичности структур тех или иных сложных систем. В качестве примеров можно назвать транспортные, энергетические системы, системы связи, электронные схемы, системы распознавания образов, а также задачи математической химии, исследование социальных сетей и многие другие.

Поскольку для данной задачи, с одной стороны, не построен алгоритм решения, имеющий полиномиальную трудоемкость, с другой – не доказана NP-полнота, многие авторы занимаются разработкой и исследованием новых алгоритмов. При этом исследуются разные постановки задачи, в том числе для ориентированных графов [2], однако в большинстве случаев задача рассматривается для неориентированных графов без петель и кратных ребер.

Не так давно Л. Бабай [3] предложил алгоритм решения задачи, имеющий квазиполиномиальную трудоемкость $exp((\log n)^{O(1)})$. Вместе с тем алгоритм весьма сложен для понимания и его корректность, насколько нам известно, на момент написания статьи не была подтверждена. В большинстве случаев авторы при решении задачи используют понятие инварианта [4], то есть некоторой количественной характеристики структуры графа, которая остается неизменной при перенумерации его вершин. В качестве примеров работ, предлагающих алгоритмы решения задачи и их программную реализацию можно назвать [5—10]. В числе прочих предлагаются алгоритмы, которые за полиномиальное время, если и не решают задачу полностью, то вычисляют некоторые характеристики, которые могут быть использованы для решения полной задачи [11, 12].

Отметим, что чаще всего авторы ищут решение задачи в постановке из [1], то есть их алгоритм должен просто ответить на вопрос, изоморфны графы или нет. Обычно решение сводится к попытке построения полных инвариантов обоих графов, сравнение которых и дает ответ на поставленный вопрос. Вместе с тем для практических задач не менее важно построить подстановку, задающую соответствие между вершинами двух графов. Поэтому мы в своей работе будем решать именно эту задачу, тем более, что имея такую подстановку мы сразу можем проверить, действительно ли исходные графы изоморфны. Ввиду высокой трудоемкости решения задачи мы также ставили перед собой цель добиться существенного ускорения за счет построения параллельного алгоритма решения задачи.

В распоряжении автора были программные инструменты для организации параллельных вычислений в соответствии концепцией рекурсивно-параллельного (РП) программирования. Основные принципы организации рекурсивно-параллельных вычислений, и основные алгоритмы и механизмы поддержки этого стиля программирования описаны в [13]. Разработанные автором библиотеки [14, 15] позволяют относительно легко создавать, отлаживать и эксплуатировать РП-приложения в среде .NET Framework. В [16] подробно описаны функциональные возможности упомянутых библиотек. Они успешно применялись при разработке и исследовании параллельных алгоритмов для решения задачи о клике [16], задачи коммивояжера [17] и задачи о рюкзаке [18].

1. Постановка задачи

Напомним формулировку задачи. Пусть есть два неориентированных графа, заданных своими множествами вершин и ребер: $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$. Требуется найти функцию $f(V_1 \longrightarrow V_2)$, такую что $\{u, v\} \in E_1 \iff \{f(u), f(v)\} \in E_2$, либо доказать ее отсутствие. Напомним, что как было отмечено выше, нашей целью является нахождение этой функции, задающей подстановку, или доказательство ее отсутствия.

2. Последовательный алгоритм решения задачи

Сначала опишем предлагаемый последовательный алгоритм, поскольку именно он является основой для дальнейшего распараллеливания решения задачи.

Ключевым понятием при построении алгоритма является инвариант вершины $I(v)$ – характеристика, которая позволяет утверждать, что если $v_1 \in V_1$, $v_2 \in V_2$ и $I(v_1) \neq I(v_2)$, то в искомой подстановке $f(v_1) \neq v_2$. Понятие инварианта вершины очень часто используется для решения задачи об изоморфизме, причем различными авторами предлагаются самые разные варианты их задания [5, 11, 12].

В качестве возможных значений инвариантов вершин при построении своего алгоритма мы рассматривали следующие:

1. Массив $L(v) = \{l_i\}$, $i \in \{1, ..., d\}$, где d – диаметр компоненты графа, к которой принадлежит v, l_i – количество вершин, отстоящих от v на расстояние i.
2. Множество из двух массивов $\{L(v), M(v)\}$, где $L(v)$ имеет тот же смысл, что и в предыдущем варианте, а $M(v) = \{m_i\}$, $i \in \{1, ..., d\}$, где m_i – количество ребер, связывающих между собой вершины, отстоящие от v на расстояние i.
3. Множество из трех массивов $\{L(v), M(v), N(v)\}$. Здесь $L(v)$ и $M(v)$ имеют тот же смысл, что и в предыдущем варианте, а $N(v) = \{n_i\}$, $i \in \{1, ..., d-1\}$, где n_i – количество ребер, связывающих между собой вершины, отстоящие от v на расстояние i с вершинами, находящимися на расстоянии $i + 1$ от вершины v.

Вычисление инвариантов для всех вершин обоих графов было первым шагом нашего алгоритма. Трудоемкость этого этапа, очевидно, оценивается как $O(n^3)$. Отметим сразу, что в ходе эксперимента мы отдали предпочтение второму варианту, он требовал по сравнению с первым в полтора-два раза больше времени, но очень сильно сокращал последующие вычисления. Третий вариант, казалось бы, позволяет более детально характеризовать вершины, однако по сравнению со вторым существенного ускорения на последующих этапах решения задачи не обеспечивал.

Перейдем к описанию базового последовательного алгоритма. При этом мы будем использовать понятия частичной подстановки и перестановки.

Частичная подстановка S представляет собой массив длины n, в котором часть элементов имеет неопределенное значение (по умолчанию), остальным присвоены значения $S[i] = j$, соответствующие значениям функции $f(i) = j$. Подстановка, заполненная целиком, определяет искомую подстановку. Перестановка T представляет собой массив из k номеров вершин. В процессе перебора подстановок нам требуется перебрать все $k!$ вариантов их расположения. Матрицы смежности для первого и второго графа мы обозначим A_1 и A_2.

Предлагаемый алгоритм последовательно выполняет следующие действия:

1. Создаем массивы инвариантов всех вершин обоих графов. При этом одновременно вычисляются диаметры графов (для статистики) и соответственно определяется их связность.
2. Если графы оказались несвязными, разбиваем их на компоненты и дополняем инварианты для каждой вершины указанием на то, к какой компоненте связности она принадлежит. Номера компонент связности для разных графов совпадать не обязаны, они нужны только для того, чтобы после сортировки вершин и разбиения их по группам в каждой группе присутствовали вершины из одной компоненты связности.
3. Сортируем массивы инвариантов вершин для каждого из графов. Отношение порядка между двумя инвариантами задается самым естественным образом – лексикографически.
4. Определяем группы вершин с одинаковыми инвариантами и их размеры. В результате проведенной ранее сортировки группы вершин, потенциально соответствующих друг другу, располагаются в одинаковом порядке для обоих графов. В рамках одной группы, конечно, вершины могут находиться в произвольном порядке.

5. Проверяем на совпадение размеров групп. Если они не совпадают, то графы, очевидно, не изоморфны.
6. Пытаемся максимально измельчить группы. Для этого строим и проверяем частичную подстановку S, которая задает значения искомой подстановки f (пока еще не для всех вершин). В нее включаем все вершины из групп единичного размера – их соответствие друг другу в искомой подстановке (если графы изоморфны) не вызывает сомнений. Каждую из вновь добавленных вершин i проверяем на выполнение равенства $A_2[S[i], S[j]] = A_1[i, j]$, которое должно выполняться для всех вершин j, включенных в частичную перестановку на этом и более ранних этапах.
7. Далее в цикле пробуем сделать группы с одинаковыми инвариантами еще мельче с учетом текущей частичной подстановки S. Для этого вершины в пределах одной группы (они имеют одинаковые инварианты) сортируются (лексикографически) по значению вектора из нулей и единиц, которые указывают на наличие ребра между данной вершиной и вершинами, уже включенными в частичную подстановку. Затем происходит дробление ранее созданных групп на более мелкие с использованием того же способа сравнения. После очередного уменьшения размеров групп выделяем вновь появившиеся группы единичного размера, проверяем измельченные группы на совпадение размеров и в случае совпадения достраиваем текущую частичную подстановку за счет новых групп единичного размера. Цикл завершается, если в результате очередной итерации не появилось новых групп единичного размера.
8. Окончательно решаем задачу перебором оставшихся вариантов подстановок (рекурсивно). Естественно, подстановки перебираются только для идентичных групп вершин в обоих графах, это многократно уменьшает количество вариантов для рассмотрения. Блок-схема алгоритма этого этапа вычислений приводится ниже.

Рассмотрим алгоритм перебора оставшихся вариантов подстановок. Строго говоря, если осталось m групп размера $k_1, k_2, ..., k_m$, количество подстановок, который потребуется перебрать, равно $K = k_1! * k_2! * ... * k_m!$, однако предлагаемый алгоритм на каждой ветке отсекает огромное количество неподходящих подстановок, в результате вычисления заканчиваются очень быстро. В ходе экспериментов, даже если изначально величина K имела порядок $10^{50} – 10^{100}$, на деле требовалось перебрать несколько тысяч или десятков тысяч подстановок, что на современных компьютерах выполняется очень быстро. Зачастую эта величина измерялась десятками или даже единицами.

Блок-схема рекурсивной функции $RecursiveBruteForce(rL)$, используемой для окончательного перебора подстановок приведена на рисунке 1. В ней требуют пояснений некоторые действия, записанные в третьем блоке. Основной причиной радикального замедления работы может быть ситуация, когда в одной группе вершин (напомним, что они принадлежат одной компоненте связности) все вершины имеют одинаковые инварианты, например, образуют кольцо. Они, конечно, могут быть пронумерованы в произвольном порядке, но их требуется сопоставить вершинам идентичной группы во втором графе. А для этого их требуется выстроить по порядку следования в кольце, чтобы не перебирать все варианты, которых может быть чрезвычайно много. Необходимость включения этого блока в алгоритм выяснилась при тестировании программы на регулярных графах со степенью вершины 2. Разумеется, могут встретиться и другие особенные случаи, например, компоненты, представляющие собой полные двудольные графы. Впрочем, в этом случае можно просто инвертировать граф и получить два кольца. Разумеется, такого рода примеров можно построить много, но все они будут носить искусственный характер, предусмотреть все возможные варианты невозможно. Кольца же нередко могут встретиться в обычных ситуациях, если графы, с которыми мы имеем дело, имеют низкую плотность.

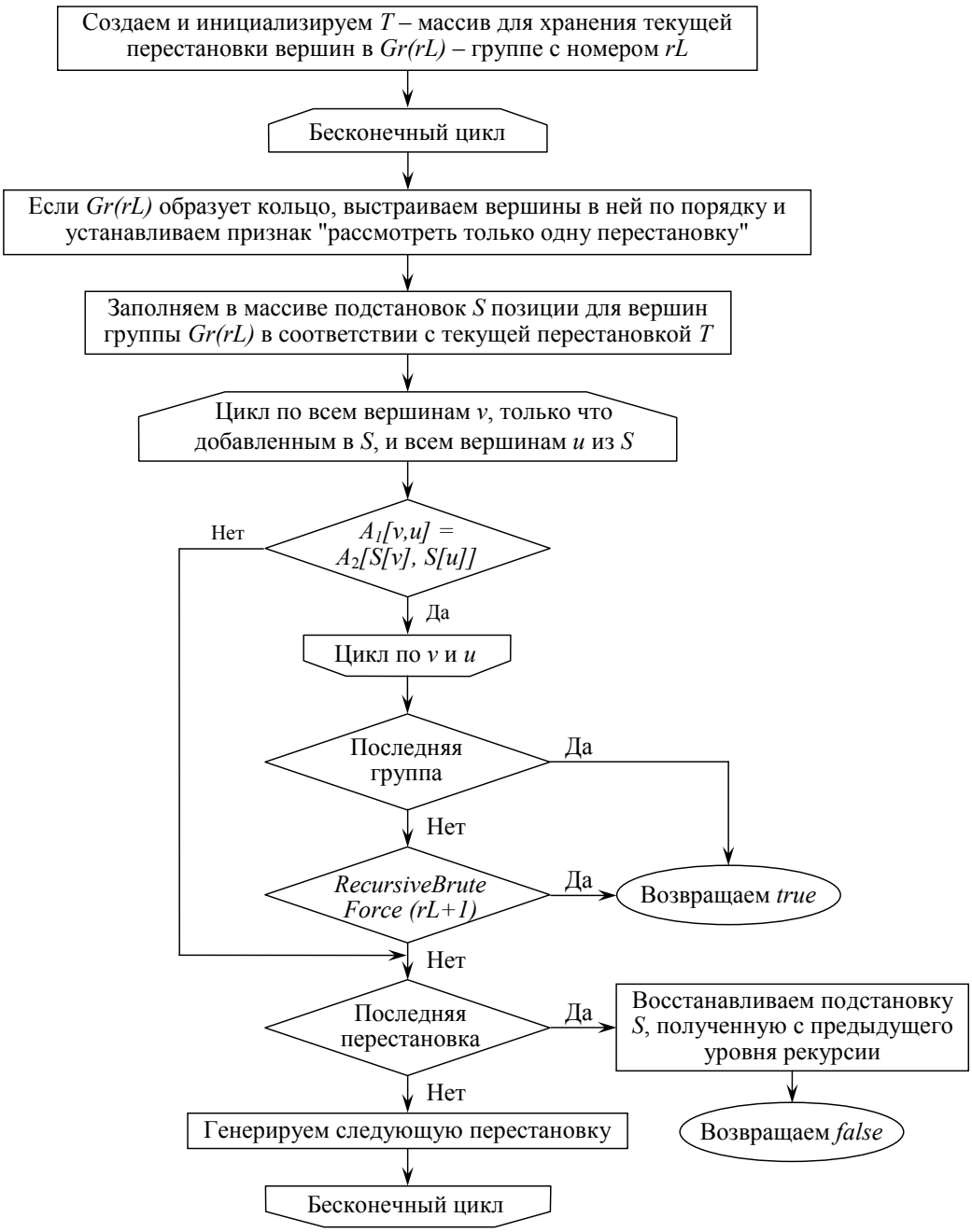

Fig. 1. Block diagram of the function *RecursiveBruteForce(rL)* to enumerate substitutions

Рис. 1. Блок-схема функции перебора подстановок *RecursiveBruteForce(rL)*

3. Распараллеливание алгоритма

В описанном выше последовательном алгоритме основными претендентами на распараллеливание являются первый и последний шаги, поскольку трудоемкость остальных, очевидно, намного ниже, и организация параллельных вычислений для них существенного выигрыша дать не может.

Для первого этапа (вычисление инвариантов для всех вершин обоих графов) распараллеливание легко может быть осуществлено в соответствии со стандартной РП-схемой [13]. Основным параметром каждой задачи является количество вершин, для которых нужно вычислить инварианты. Это количество делится пополам, подзадача для вычисления одной половины остается для решения на данном процессорном модуле (ПМ), другая – оформляется как потенциально мигрирующий процесс и, если необходимо, может быть передана для исполнения на другом ПМ. Библиотека поддержки рекурсивно-параллельного стиля программирования [16] обеспечивает достаточно равномерное распределение работы по системе на начальном этапе вычислений и при необходимости динамическое его перераспределение на последующих этапах.

Автор изначально рассматривал возможность распараллеливания последнего шага алгоритма (окончательный перебор подстановок), однако затем отказался от этой идеи, хотя построение такого РП-алгоритма не представляло сложности. Причина в том, что в ходе экспериментов выяснилось, что несмотря на отсутствие полиномиальной оценки сложности, этот этап не требовал слишком большого количества вычислений. Качество разбиения вершин на группы и эффективность отсечения неперспективных вариантов описанного выше алгоритма во всех экспериментах приводили к очень быстрому получению искомого результата. При таких условиях организация параллельных вычислений могла привести только к замедлению работы из-за неизбежных накладных расходов на порождение и запуск параллельных активаций процедур.

4. Описание эксперимента и его результаты

Исходные данные для тестирования генерировались случайным образом. Для усложнения задачи было принято решение исследовать работу алгоритма на регулярных графах. Было сгенерировано несколько десятков графов с количеством вершин 7000 и степенью вершины от 2 до 3000. Графы с плотностью более 0.5 не использовались для тестирования, поскольку их можно просто инвертировать и исследовать получившиеся графы. Изначально для генерации случайных регулярных графов использовался алгоритм, предложенный в [19], однако для такого количества вершин он работал слишком медленно, и автором был разработан свой, существенно более быстрый алгоритм.

В процессе тестирования использовались компьютеры на базе двухядерного процессора Intel Core i3-7100 (максимальное количество потоков 4) с тактовой частотой 3.90 GHz и 8 GB оперативной памяти, работающие под управлением 64-разрядной ОС Windows 10. Пропускная способность сети равнялась 100 Mb/s.

В таблице 1 приведены некоторые результаты вычислений, позволяющие оценить время решения задачи последовательным алгоритмом и долю вычислений, приходящихся на первый этап (вычисление инвариантов вершин). Мы показали результаты не для всех рассмотренных вариантов исходных данных, а только небольшую их часть, однако они все похожи друг на друга, слегка выделяются только результаты тестирования для графов со степенью вершины, равной 2, они по вполне понятным причинам оказались несвязными.

Для этих же графов мы провели эксперимент по оценке эффективности параллельного алгоритма. В таблице 2 приведены результаты, демонстрирующие ускорение параллельного алгоритма по отношению к последовательному для количества задействованных компьютеров (ПМ) до 16. Показано как ускорение собственно вычислений, так и ускорение с учетом необходимости передачи по сети исходных данных и вычисленных результатов.

Table 1. Execution time of sequential algorithm for regular graphs on 7000 vertices

Таблица 1. Длительность работы последовательного алгоритма для регулярных графов на 7000 вершин

Степень графа	**2**	**3**	**5**	**20**	**200**	**300**
Диаметр графа	∞	16	9	4	3	2
Все вычисления (с)	*941.11*	*3149.81*	*3240.97*	*2381.29*	*2794.24*	*2243.00*
Выч. инвариантов	932.41	3149.22	3240.36	2380.51	2792.84	2241.73
Инварианты, %	*99.08*	*99.98*	*99.98*	*99.97*	*99.95*	*99.94*

Table 2. Speed increase for regular graphs on 7000 vertices

Таблица 2. Ускорение вычислений для регулярных графов на 7000 вершин

Кол-во ПМ	Степень графа	***2***	***3***	***5***	***20***	***200***	***3000***
	Диаметр графа	∞	16	9	4	3	2
1	Уск. вычислений	2.17	2.41	2.38	2.47	2.63	2.84
	Уск. всей работы	*2.17*	*2.41*	*2.38*	*2.47*	*2.63*	*2.84*
2	Уск. вычислений	4.27	4.80	4.69	4.89	5.25	5.59
	Уск. всей работы	*4.13*	*4.76*	*4.66*	*4.85*	*5.20*	*5.52*
4	Уск. вычислений	8.03	9.47	9.37	9.61	9.88	10.81
	Уск. всей работы	*7.59*	*9.34*	*9.24*	*9.43*	*9.72*	*10.53*
6	Уск. вычислений	11.15	13.57	13.47	13.47	14.34	15.39
	Уск. всей работы	*10.33*	*13.29*	*13.20*	*13.11*	*14.01*	*14.82*
8	Уск. вычислений	12.75	15.91	15.43	16.16	17.07	18.61
	Уск. всей работы	*11.34*	*15.33*	*14.86*	*15.43*	*16.32*	*17.39*
12	Уск. вычислений	19.29	25.28	24.83	24.55	26.28	27.66
	Уск. всей работы	*16.80*	*24.23*	*23.87*	*23.26*	*25.05*	*25.69*
16	Уск. вычислений	22.20	31.67	30.55	32.64	33.08	34.96
	Уск. всей работы	*16.81*	*27.91*	*27.38*	*28.21*	*28.94*	*30.05*

Результаты наглядно показывают хорошие перспективы для ускорения алгоритма за счет его распараллеливания. Тот факт, что даже на одном компьютере параллельная версия программы работает быстрее, чем последовательная, объясняется тем, что она задействует для вычислений несколько потоков, что дает ощутимое ускорение на многоядерном процессоре.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co, San Francisco, 1979.

[2] D. C. Schmidt and L. E. Druffel, "A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices", *Journal of the ACM (JACM)*, vol. 23, no. 3, pp. 433–445, 1976.

[3] L. Babai, "Graph isomorphism in quasipolynomial time", in *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 2016, pp. 684–697.

[4] F. Harary, *Graph theory*. Addison-Wesley, 1969.

[5] Y. German, O. German, and A. Dunaev, "An algorithm for establishing graph's isomorfism", *Proceedings of BSTU. Issue 3, Physics and mathematics. Informatics*, no. 2 (200), pp. 114–117, 2017.

[6] V. K. Pogrebnoy and A. Pogrebnoy, "Polynomial algorithm of computing complete graph invariant on the basis of integral structure descriptor", *Bulletin of the Tomsk Polytechnic University*, vol. 323, no. 5, pp. 152–159, 2013.

[7] V. K. Pogrebnoy and A. Pogrebnoy, "Polynomiality of method for computing graph structure integral descriptor", *Bulletin of the Tomsk Polytechnic University*, vol. 323, no. 5, pp. 146–151, 2013.

[8] A. Pogrebnoy, "Complete graph invariant and algorithm of its computation", *Bulletin of the Tomsk Polytechnic University*, vol. 325, no. 5, pp. 110–122, 2014.

[9] A. Pogrebnoy and V. K. Pogrebnoy, "Method of graph vertices differentiation and solution of the isomorphism problem", *Bulletin of the Tomsk Polytechnic University*, vol. 326, no. 6, pp. 34–45, 2015.

[10] A. Pogrebnoy and V. K. Pogrebnoy, "Method of graph vertices differentiation and solution of the isomorphism problem in geoinformatics", *Bulletin of the Tomsk Polytechnic University*, vol. 326, no. 11, pp. 56–66, 2015.

[11] B. F. Melnikov and N. P. Churikova, "Algorithms of Comparative Analysis of Two Invariants of a Graph", *Sovremennye informacionnye tehnologii i IT-obrazovanie (Modern Information Technologies and IT-Education)*, vol. 15, no. 1, pp. 45–51, 2019.

[12] G. S. Ivanova and V. A. Ovchinnikov, "Completely described undirected graph structure", *Science and Education of the Bauman MSTU*, no. 4, pp. 106–123, 2016.

[13] V. V. Vasilchikov, *Sredstva parallelnogo programmirovaniya dlya vychislitelnykh sistem s dinamicheskoy balansirovkoy zagruzki*. YarGU, Yaroslavl, 2001.

[14] V. V. Vasilchikov, "Kommunikatsionnyy modul dlya organizatsii polnosvyaznogo soedineniya kompyuterov v lokalnoy seti s ispolzovaniem .NET Framework", *Svidetelstvo o gosudarstvennoy registratsii programmy dlya EVM № 2013619925*, 2013.

[15] V. V. Vasilchikov, "Biblioteka podderzhki rekursivno-parallelnogo programmirovaniya dlya .NET Framework", *Svidetelstvo o gosudarstvennoy registratsii programmy dlya EVM № 2013619926*, 2013.

[16] V. V. Vasilchikov, "On the recursive-parallel programming for the. NET framework", *Automatic Control and Computer Sciences*, vol. 48, no. 7, pp. 575–580, 2014.

[17] V. V. Vasilchikov, "On optimization and parallelization of the little algorithm for solving the travelling salesman problem", *Automatic Control and Computer Sciences*, vol. 51, no. 7, pp. 551–557, 2017.

[18] V. V. Vasilchikov, "On a recursive-parallel algorithm for solving the knapsack problem", *Automatic Control and Computer Sciences*, vol. 52, no. 7, pp. 810–816, 2018.

[19] A. Steger and N. Wormald, "Generating random regular graphs quickly", *Combinatorics, Probability and Computing*, vol. 8, no. 4, pp. 377–396, 1999.